

**Software and Automation Systems Branch  
C++ Style Guide  
Version 2.0**

**APPROVAL**

---

Sylvia B. Sheppard Head, Software and Automations Systems Branch Code 522	Date
---------------------------------------------------------------------------------	------

---

Julie D. Breed Head, Systems and Software Environments Section Code 522	Date
-------------------------------------------------------------------------------	------

Goddard Space Flight Center  
Greenbelt, Maryland

## PREFACE

This document describes the Code 522/Goddard Space Flight Center (GSFC) Software and Automation Systems Branch C++ Style Guide, Version 2.0. This is a major update to the "Software and Automation Systems Branch's C++ Style Guide Version 1.0," Edwards, S. Et. Al., (DSTL-92-009). The "Software Engineering Branch's C Style Guide," Doland, J. Et. Al., (SEL-94-003) and the draft of the "Software Engineering Branch's C++ Style Guide," Shoan, W. Et. Al., June 1995 were used extensively as a basis in development of this document.

This document also discusses recommended practices and style for programmers using the C++ language in the Software and Automation Systems Branch, Code 522. Guidelines are based on generally recommended software engineering techniques, industry resources, and local convention.

This document is under the configuration management of the Software and Automation Systems Branch Configuration Control Board (CCB). Changes to this document shall be made by Documentation Change Notice (DCN), reflected in text by change bars, or by complete revision.

Requests for copies of this document, along with questions and proposed changes, should be addressed to:

Technology Support Office  
Software and Automation Systems Branch, Code 522  
Goddard Space Flight Center  
Greenbelt, Maryland 20771

**CHANGE INFORMATION PAGE**

<i>List of Effective Pages</i>		
<b>Page Number</b>	<b>Issue</b>	
Title	Original	
Signature Page	Original	
iii through x	Original	
1-1	Original	
2-1 and 2-8	Original	
3-1 through 3-3	Original	
4-1 through 4-12	Original	
51 through 5-10	Original	
6-1 and 6-3	Original	
A-1 through A-8	Original	
B-1	Original	
<i>Document History</i>		
<b>Issue</b>	<b>Date</b>	<b>DCN No.</b>
Original	June 1996	N/A

## **ACKNOWLEDGMENTS**

The following individuals were developers of the guidelines, who gave generously of their time and expertise during the process of development:

Julie Breed  
Jeremy Jones  
Johnny Medina  
Walt Moleski  
Greg Shirah  
Dana Uehling  
Susan Valett  
Bob Wiegand

## TABLE OF CONTENTS

<b>SECTION 1 - INTRODUCTION.....</b>	<b>1-1</b>
1.1 Purpose .....	1-1
1.2 Audience.....	1-1
1.3 Document Format.....	1-1
<b>SECTION 2 - CONSISTENT STYLE.....</b>	<b>2-1</b>
2.1 White Space.....	2-1
2.1.1 Blank Lines.....	2-1
2.1.2 Spacing.....	2-2
2.1.3 Indentation .....	2-3
2.1.4 Continuation Lines .....	2-3
2.2 Comments.....	2-5
2.3 Standard Naming Conventions .....	2-7
2.3.1 Name Formats .....	2-7
2.3.2 Short Names .....	2-8
2.3.3 General Guidelines for Variable Names.....	2-8
<b>SECTION 3 - PROGRAM ORGANIZATION.....</b>	<b>3-1</b>
3.1 Program Files .....	3-1
3.2 Encapsulation and Information Hiding .....	3-1
3.3 README File.....	3-1
3.4 Header Files.....	3-2
3.5 Source Files .....	3-3
3.6 Makefiles .....	3-3
<b>SECTION 4 - FILE ORGANIZATION.....</b>	<b>4-1</b>
4.1 File Prolog .....	4-1
4.2 Header File (Class Definition) (*.h).....	4-2
4.2.1 Order of Contents .....	4-2
4.2.2 Format of Class Declarations.....	4-2
4.2.3 Method / Function Documentation in Header File .....	4-2
4.2.4 Sample Header File .....	4-3
4.3 Source File (Class Implementation) (*.cc).....	4-4
4.3.1 Order of Contents .....	4-4
4.3.2 Method Function Implementation and Documentation in Source File .....	4-4
4.3.3 Sample Source File.....	4-6
<b>SECTION 5 - USE OF LANGUAGE CONSTRUCTS.....</b>	<b>5-1</b>
5.1 Functions .....	5-1
5.2 Variables.....	5-1
5.3 Constants .....	5-2
5.4 Pointers and Structures.....	5-3
5.5 Type Conversions and Casts.....	5-3
5.6 Use of Operators.....	5-5
5.7 Assignment Operators and Initialization.....	5-5
5.8 Conditional Expressions .....	5-6
5.9 Selection Control Statements (if-then-else) .....	5-7
5.10 Switch Statements .....	5-9
5.11 Iteration Control Statements .....	5-9
5.12 Goto's and Labels .....	5-10
5.13 Memory Allocation .....	5-10
5.14 Standard Typedef Scheme .....	5-10

**TABLE OF CONTENTS (CONT'D)**

<b>SECTION 6 - DESIGN AND IMPLEMENTATION.....</b>	<b>6-1</b>
6.1 Guidelines for Portability.....	6-1
6.2 Guidelines for Performance.....	6-2
6.3 Program Design Language.....	6-2
6.4 Compiler Warnings.....	6-3
<b>APPENDIX A .....</b>	<b>A-1</b>
<b>BIBLIOGRAPHY .....</b>	<b>B-1</b>

**EXAMPLES**

Example 2.1.1a - Code Paragraphing.....	2-1
Example 2.1.2a - Proper Spacing.....	2-2
Example 2.1.2b - Poor Spacing.....	2-2
Example 2.1.4a - Strings of Conditional Operators.....	2-3
Example 2.1.4b - Function Call Method.....	2-3
Example 2.1.5a - Braces-Stand-Alone Method.....	2-4
Example 2.1.5b - Braces Improve Readability.....	2-4
Example 2.1.5c - No Braces - Difficult to Read.....	2-4
Example 2.1.5d - Comment Closing Braces.....	2-5
Example 2.1.5e - Dummy Body.....	2-5
Example 2.2a - Boxed Comment.....	2-5
Example 2.2b - Section Separator.....	2-6
Example 2.2c - Block Comment.....	2-6
Example 2.2d - In-Line Comments.....	2-6
Example 2.2e - Block Comment vs. In-Line Comment.....	2-7
Example 2.2f - Comment Indentation.....	2-7
Example 5.3a - Potential Problems with Using #define.....	5-2
Example 5.3b - Using the 'const' Modifier is Preferred.....	5-2
Example 5.3c - Enumeration Types.....	5-2
Example 5.5a - Casting Gets Buggy Code by Compiler.....	5-3
Example 5.5c - Acceptable Use of Casting.....	5-4
Example 5.7a - Embedded Assignment.....	5-5
Example 5.7b - Embedded Assignment (not recommended).....	5-6
Example 5.7c - Recommended Embedded Assignment (recommended).....	5-6
Example 5.8a - Simple Conditional Expression.....	5-6
Example 5.8b - Complex Conditional Expression (not recommended).....	5-6
Example 5.9a - Nested 'if' Statements.....	5-7
Example 5.9b - Use of Braces Result (recommended).....	5-8
Example 6.1a - Centrally Placed Definitions.....	6-1
Example 6.3a - Understandable PDL.....	6-3

## SECTION 1 INTRODUCTION

### 1.1 PURPOSE

This document describes the style recommended by the Software and Automation Systems Branch for writing C++ programs, where the goals are to produce code that is:

- a. Reliable
- b. Maintainable
  - 1. Organized
  - 2. Easy to understand
  - 3. Portable
- c. Efficient

### 1.2 AUDIENCE

This document was written specifically for programmers in the Software and Automation Systems Branch, Code 522 environment. This document assumes a working knowledge of C++, and focuses on pointing out good practices that will enhance the quality of the C++ code.

### 1.3 DOCUMENT FORMAT

The following meanings are assigned to the formats included in this document:

Example code is included in shaded boxes.

Plain boxes are used to enclose standard formats and other related information.

*Italics are used to differentiate the variable portion to be completed by the programmer from the standard format that should appear verbatim.*

*Italics are also used to visually separate brief (one- line) examples.*

**Bold** text is used simply to highlight key words or phrases for the reader.

## SECTION 2

### CONSISTENT STYLE

The guidelines in this section are written to improve the consistency of C++ programming style within the Branch. This will ease the job of maintenance, and will also make it easier for a programmer to transfer from one project to another.

#### 2.1 WHITE SPACE

Adding **white space** in the form of blank lines, spaces, and indentation significantly improves the readability of code.

##### 2.1.1 BLANK LINES

A careful use of **blank lines** between code "paragraphs" can greatly enhance readability by making the logical structure of a sequence of lines more obvious. Using blank lines to create paragraphs in code or comments can make programs more understandable. The following example illustrates how the use of blank lines helps break up lines of text into meaningful portions.

Example 2.1.1a - Code Paragraphing

```
bool Foo::Split (const DstdString& data,          //      string to split
                const char delimiter,            //      the character to break on
                DstdString& first,               //      return string up to delimiter
                DstdString& remainder)           //      return string after delimiter
{
    const char* raw = (const char*) data;

    int p = 0;
    while (raw[p] && raw[p] != delimiter)
    {
        ++p;
    }

    first = data(0, p-1);
    remainder = data(p+1, data.Length());

    return (remainder.Length() > 0 ? true : false);
}
```

However, overuse of blank lines can defeat the purpose of grouping and can actually reduce readability. Therefore, a single blank line should be used to separate sections of a program from one another.

## 2.1.2 SPACING

Appropriate **spacing** enhances the readability of lexical elements.

- a. Do not put space around the **primary operators**: `->`, `..`, and `[]`:  
`p->m    s.m a[i]`
- b. Do not put a space before **parentheses** following function names.  
`exp(2, x)`
- c. Do not put spaces between **unary operators** and their operands:  
`!p   -b   ++i   -n   *p   &x`
- d. **Casts** are the only exception. *Do put a space* between a cast and its operand:  
`(long) m`
- e. Always put spaces around **assignment operators**:  
`c1 = c2`
- f. Always put space around **conditional operators**:  
`z = (a > b) ? a : b;`
- g. **Commas** should have one space (or a new line) after them:  
`strncat(t, s, n)`
- h. **Semicolons** should have one space (or a new line) after them:  
`for (i = 0; i < n; ++i)`
- i. For **other operators**, generally put one space on either side of the operator:  
`x + y   a < b   && b < c`

The following examples illustrate how to use individual spaces to improve readability and to avoid errors. The second example is not only harder to read, but the spacing introduces an error, where the operator `/*` will be interpreted by the compiler as the beginning of a comment.

Example 2.1.2a - Proper Spacing

```
*average = *total / *count;
```

Example 2.1.2b - Poor Spacing

```
*average=*total/*count;
      ^begin comment that has no end
```

### 2.1.3 INDENTATION

Indentation should be used to show the logical structure of code. Research has shown that **four spaces** are the optimum indent for readability and maintainability. Tabs should be used for indentation and it is recommended that they be set to four spaces for displays. However, in highly nested code with long variable names, four space indentation may cause a line of code to overrun at the end of the line. So, indentation of four spaces is recommended unless other circumstances make it unworkable.

### 2.1.4 CONTINUATION LINES

Statements that continue over more than one line should be indented each line after the first line with **two additional tabs**. This will differentiate the continuation portion of the statement from the encapsulated body of a block.

- a. Strings of conditional operators that will not fit on one line should be divided into separate lines, breaking before the logical operators in the following format:

Example 2.1.4a - Strings of Conditional Operators

```
if (fAlive
    && (fSkipValidation || command.Valid())
    && fCommandProcessor.Ready())
{
    SaveState();
    fCommandProcessor.Execute(command);
    Log(command);
}
```

- b. Function calls that continue over more than one line should be presented one argument per line, followed by a comma in the following format. The benefit of this is that an in-line comment can be added after each argument to describe its purpose. If argument names are short, and their meanings are simple, they can be combined on one line at the programmer's discretion.

Example 2.1.4b - Function Call Method

```
fTimer = new WrTimeEvent<Foo, void (Foo::*)(WrTimeEventBase*)>(
    fContext,                // context of event to time
    fInterval,               // timing interval
    *this,                   // current interval
    &Foo::TimerHandler);     // routine to process timer
```

## 2.1.5 BRACES AND PARENTHESES

Compound statements, also known as blocks, are lists of statements enclosed in braces. The recommended brace style is the **Braces-Stand-Alone** method. Braces should be placed on separate lines and aligned. This style allows for easier pairing of the braces and costs only one vertical space.

Example 2.1.5a - Braces-Stand-Alone Method

```
int total = 0;
for (int i=0; i<elements; i++)
{
    total += i;
    sum[i] = total;
}
```

- a. Although C++ does not require braces around single statements, braces should be used for single statement blocks to help improve the readability and maintainability of the code. If braces are not used in single statement blocks the risk of maintenance errors is increased. Maintenance programmers may add a statement within the block, but may forget to add braces.
- b. Nested conditionals and loops can often benefit from the addition of braces, especially when a conditional expression is long and complex. The following examples show the same code with and without braces. The use of braces is encouraged to improve readability.

Example 2.1.5b - Braces Improve Readability

```
for (dp = &values[0]; dp < top_value; dp++)
{
    if ((dp->dValue == argValue)
        && ((dp->dFlag & argFlag) != 0))
    {
        return (dp);
    }
}
return (NULL);
```

Example 2.1.5c - No Braces - Difficult to Read

```
for (dp = &values[0]; dp < topValue; dp++)
    if ((dp->dValue == argValue)
        && ((dp->dFlag & argFlag) != 0))
        return (dp);
return (NULL);
```

- c. If the span of a block is large (more than about 40 lines) or there are several nested blocks, **comment closing braces** to indicate what part of the process they delimit:

Example 2.1.5d - Comment Closing Braces

```
SymbolTableIter iter(fSymbolTable);
for (Symbol symbol = iter.First(); iter.More(); symbol = iter.Next())
{
    if (symbol.Defined())
    {
        ...
    }    //      defined
else
    {
        ...
    }    //      not defined
}    //      for all symbols
```

- d. If a 'for' or 'while' statement has a dummy body, the semicolon should go on the next line. It is good practice to add a comment stating that the dummy body is deliberate.

Example 2.1.5e - Dummy Body

```
while (fTable.LoadElement())
;    //      dummy body
```

- e. Insert a space between reserved words and their opening parentheses.
- f. Insert parentheses around the objects of size of and return.

## 2.2 COMMENTS

Comments in the code can provide information that a person could not discern simply by reading the code. In addition to the standard comment blocks required to appear (see section 4), comments can be added at many different levels. Comments can be written in several styles depending on their purpose and length. Use comments to **add information** for the reader or to **highlight sections** of code. Comments should not paraphrase the code.

There are several different formats of comments:

- a. **Boxed Comments** - Used for standard format comment blocks (see section 4).

Example 2.2a - Boxed Comment

```
//=====
=
//This code was developed by NASA, Goddard Space Flight Center, Code 520
// for the Project Name Project
//
//
//=====
=
```

- b. **Section Separators** - Used to visually separate portions of a program or comment block.

Example 2.2b - Section Separator

```
//-----  
-
```

- c. **Block comments** - Used at the beginning of each major section of the code as a narrative description of that portion of the code.

Example 2.2c - Block Comment

```
// This is a block comment. The comment should be written in full sentences,  
// with correct punctuation, etc. Use this form of comment when more than one  
// sentence is required.
```

- d. **In-Line Comments** - Written on the same line as the code or data definition they describe. These comments should be tabbed over far enough to separate them from the code statements. In addition, if more than one short comment appears in a block of code or data definition, they should all be started from the same tab position.

Example 2.2d - In-Line Comments

```
Matrix33      transformation Matrix;      // matrix converting from GCI to BCS  
unsigned int   n;                          // number of attitude models used  
String        name;                        // name of current attitude model
```

General guidelines for commenting are as follows:

- Use in-line comments to document variable definitions and block comments to describe computation processes (see Example 2.2e).
- Where data are being declared or defined, in-line comments should be added to explain the purpose of the variables.
- The C++ comment indicator (“//”) should be used exclusively (as opposed to the alternatives such as /\*).
- Comments should describe blocks of code rather than individual lines of code (see Example 2.20).
- Comments should be written at the same level of indentation as the code they describe (see Example 2.2f).

## Example 2.2e - Block Comment vs. In-Line Comment

preferred style:

```
//
//  Main sequence: get and process all user requests
//

while (!finish())
{
    inquire();
    process();
}
```

not recommended:

```
while (!finish())           //  Main sequence:
{                           //
    inquire();              //  Get user request
    process();              //  And carry it out
}                           //  As long as possible
```

## Example 2.2f - Comment Indentation

```
//
//  Main sequence: get and process all user requests
//

while (!finish())
{
    inquire();
    if (requestcode !=0)
    {
        //  If the request code is non-zero, then perform
        //  intermediate processing to generate request information
        generateRequestInfo(requestCode);
    }
    process();
}
```

## 2.3 STANDARD NAMING CONVENTIONS

### 2.3.1 NAME FORMATS

Classes, Types & Functions: Capitalize the first letter of each word.

*Example: CustomerSchedule*

Variables and Class Objects: Capitalize the first letter of each word except the first word.

*Example: numberOfStars*

Member Objects: Begin with the letter 'f', then conform to variable format.

*Example: fTimeOfDay*

Static Members of a Class: Begin with the letter 's', then conform to variable format.

*Example: sTimeOfDay*

Global Variables: Begin with the letter 'g', then conform to variable format.

*Example: gNumberOfStars*

Constants and Enumerators: Begin with the letter 'k', then conform to variable format.

*Example: kNumberOfStars*

Names Containing Acronyms: Follow the above conventions.

*Example: NasaIdentifier (class name)*

Names Containing Proper Nouns: Follow the above conventions.

*Example: kennedyCode (variable name)*

### 2.3.2 SHORT NAMES

Some standard **short names** for code elements are listed in the table below. While use of these names is acceptable if their meaning is clear, more explicit names are recommended.

Standard Short Names:

c	characters
i, j, k	indices
n	counters
p, q	pointers
s	strings

### 2.3.3 GENERAL GUIDELINES FOR VARIABLE NAMES

The following guidelines should be adhered to before writing code:

- Choose names with meanings that are precise and unique and use them consistently throughout the program. Longer names improve readability and clarity.
- Avoid the use of underscores.
- Avoid abbreviations. However, if required follow a uniform scheme.
- Names should differ by at least two characters. For example, "systst" and "sysstst" are easily confused.
- Do not rely on letter case to make a name unique. Although C++ is case sensitive, all names should be unique irrespective of letter case. Do not define two variables with the same spelling, but different case.
- Do not assign a variable and a typedef (or struct) with the same name, even though C++ allows this. This type of redundancy can make the program difficult to follow.
- Avoid using gratuitous modifiers (e.g., the and my) as the first word of a name.
- In separate functions, do not use identical variable names for variables that do not have the identical meaning. Using the same variable name if the meaning of two variables are only similar or coincidental can cause confusion to the reader.

## SECTION 3

### PROGRAM ORGANIZATION

This section discusses organizing program code into files. It points out good practices such as grouping logically related functions and data structures in the same file and controlling the visibility of the contents of those files.

#### 3.1 PROGRAM FILES

A C++ program consists of program files, one of which contains the `main()` function, which acts as the driver of the program. When a program is large enough to require several files, encapsulation and data hiding techniques should be used to group logically related functions and data structures into the same files. Programs should be organized as follows:

- a. A **README** file should be created to document what the program does.
- b. The **main** function should be in a separate file from all other functions.
- c. A **Makefile** should be written to make recompiles more efficient.
- d. A program should be organized into pairs of files as follows:

Class Definition: **Header File**, called *name.h*

Class Implementation: **Source File**, called *name.cc* (unless .cc is not accepted by the particular compiler)

#### 3.2 ENCAPSULATION AND INFORMATION HIDING

Encapsulation and information hiding improve the organization and maintainability of software. **Encapsulation** means grouping related elements. **Information hiding** refers to controlling the visibility (or scope) of program elements. C++ constructs can be used to control the scope of functions and data. Related information should be encapsulated in header files. Header files should be included only where needed.

A variable defined outside the current file is called an **external variable**. An external variable is only visible to a function when declared by the extern declaration, which may be used only as needed in individual functions. Extern declarations of global data should be contained in global header files so that any source that includes that header will have access to the global data as well.

#### 3.3 README FILE

A **README** file should be used to explain what the program does and how it is organized and to document issues for the program as a whole. For example, a README file might include the following:

- a. How to build and run the software.
- b. How to configure the environment.
- c. Pointers to additional documentation.

### 3.4 HEADER FILES

Header files are used to encapsulate logically related ideas. Header files may be selectively included in program files to limit visibility to only those functions that need them. They are included in implementation before compilation. Some, such as "iostream.h" are defined system wide, and must be included by any C++ program that uses the standard input/output library. Others are used with a single class or suite of classes.

- a. Header files should be used to encapsulate related definitions and declarations of functions. There should be no declarations of variables in header files, with the exception of const variables and extern declarations of global variables declared in a <name>.cc file.
- b. Greater-than/less-than brackets should be used for system include files:  
`#include <iostream.h>`  
Quotes should be used for user include files:  
`#include "ClassName.h"`  
System header files should be listed prior to user header files.
- c. Data definitions, declarations, typedefs, and enums that are needed by more than one file should be contained in a header file.
- d. Avoid using header file names that are the same as library header file names.
- e. Organize header files by class with one class definition per header file, or one group of closely related classes per file.
- f. Include preprocessor macros to avoid multiple definitions of items in a header file.
- g. The class definition should include public, protected and private sections, even if a given section is empty.
- h. Do not define a member function's implementation in the class definition unless it is necessary for performance reasons.
- i. Header files should be idempotent. That is, the file can be included more than once with the same effect as only being included exactly once.
- j. Header files should be mutually independent. That is, no header file should require that the program first include another header to work properly. The header file itself should include any other header files it needs explicitly using #include statements.
- k. Avoid unnamed parameters in declarations.

### **3.5**      **SOURCE FILES**

- a. When defining a function's implementation, it is recommended that the return type go on the same line as the function name.
- b. Functions should be defined in the order in which they appear in the class definition.
- c. External variable declarations should be avoided in the source file, as they should be declared in an included header file.
- d. If there is a private method that is not implemented, still include a prolog in the .cc file and make a note after the name that it is not implemented.
- e. If argument names are used in the source file, they must exactly match the argument names in the declaration.

### **3.6**      **MAKEFILES**

Makefiles are used on some systems to provide a mechanism for efficiently recompiling C++ code. With Makefiles, the make utility recompiles files that have been changed since the last compilation. Makefiles also allow the recompilation commands to be stored, so that potentially long commands can be greatly abbreviated. The Makefile:

- a. Lists all files that are to be included as part of the program.
- b. Contains comments documenting what files are part of libraries.
- c. Demonstrates dependencies using implicit and explicit rules.
- d. Should be well documented to aid in maintenance.

## SECTION 4 FILE ORGANIZATION

The organization of information within a file is as important to the readability and maintainability of programs as the organization of information among files. Having a consistent format throughout the branch will allow programmers to easily locate information when moving from one project to the next.

### 4.1 FILE PROLOG

This prolog should appear at the beginning of all header and source files. The format should also be used for other files related to the program, such as scripts and Makefiles, although the comment indicator ('//') must be changed.

```

/ / = = =                               F i l e                               P r o l o g
=====
//   This code was developed by NASA, Goddard Space Flight Center, Code 520 for the
//   Project Name project.
//
//--- Contents -----
//
//   Names of classes, structures, or other relevent contents
//
//--- Description -----
//
//   Description of the class written for a user of that class in the header file,
//   and written for a maintainer of that class in the source file.
//
//--- Notes -----
//
//   Anything relevant about this class, including document references, assumptions,
//   constraints, restrictions, abnormal termination conditions, etc.
//
//--- Development History -----
//
//   Date          Author          Reference
//   Description
//
//   Date          Author          Reference
//   Description
//
//   yy/mm/dd          A.Name          Reqt#xx
//   This is why I wrote the software
//
//   yy/mm/dd          B.Name          CCR#yy
//   This is why I changed the software
//
//   Etc
//
//--- Warning -----
//
//   This software is property of the National Aeronautics and Space Administration.
//   Unauthorized use or duplication of this software is strictly prohibited.
//   Authorized users are subject to the following restrictions:
//   . Neither the author, their corporation, nor NASA is responsible for
//   any consequences of the use of this software.
//   . The origin of this software must not be misrepresented either by
//   explicit claim or by omission.
//   . Altered versions of this software must be plainly marked as such.
//   . This notice may not be removed or altered.
//
/ / = = =                               E n d                               F i l e                               P r o l o g
=====

```

## **4.2        HEADER FILE (CLASS DEFINITION) (\*.h)**

### **4.2.1       ORDER OF CONTENTS**

The items contained within a header file should be presented in the following order.

```
CM tool (ie. sccs) data
#ifndef macros
#define
File Prolog (see 4.1)
#include < ... >
#include " ... "
Declarations
    Enumerations
    Typedefs
    Constants
    External Variables
    Globals
    Structures / Classes (see 4.2.2)
    Functions
#endif macros
```

### **4.2.2       FORMAT OF CLASS DECLARATIONS**

The class declarations should include public, protected, and private methods in that order, as follows.

```
Class Declaration

public:
    Declarations

protected:
    Declarations

private:
    Declarations
```

### **4.2.3       METHOD / FUNCTION DOCUMENTATION IN HEADER FILE**

In the interest of reliability and maintainability of documentation, Code 522 has decided to maintain all method/function information in the source file only. Although this goes against the object oriented philosophy by requiring users of a class to view the source file, it is felt that the benefits in terms of accuracy and singular maintenance of information are more important.

Projects may optionally choose to use a script to copy the information from the source file to the header file, perhaps only at the time of each software delivery. A sample script can be found in Appendix A.

#### 4.2.4 SAMPLE HEADER FILE

```
// %W% %G% (SCCS Control)
/   /   =   =   =                               F   i   l   e
Prolog=====
//   This code was developed for NASA, Goddard Space Flight Center, Code 520
//   for the Style Guide Project.
//
//--- Contents -----
//
//   class ExampleString
//
//--- Description -----
//
//   A generic string class.
//
//--- Notes -----
//
//   This code has been shamelessly pilfered from the DSTD library component
DstdString.
//
//--- Development History -----
//
//   96/05/13          S. Developer/522
//   Initial version created to provide style guide example.
//
//--- Warning -----
//
//   This software is property of the National Aeronautics and Space Administration.
//   Unauthorized use or duplication of this software is strictly prohibited.
//   Authorized users are subject to the following restrictions:
//   . Neither the author, their corporation, nor NASA is responsible for
//   any consequences of the use of this software.
//   . The origin of this software must not be misrepresented either by
//   explicit claim or by omission.
//   . Altered versions of this software must be plainly marked as such.
//   . This notice may not be removed or altered.
//
/ / = = =                               E n d                               F i l e                               P r o l o g
=====

#ifndef EXAMPLESTRING_H_
#define EXAMPLESTRING_H_

#include <iostream.h>

class ExampleString
{
public:

    ExampleString();
    ExampleString(const ExampleString& s);
    ExampleString(const char *p);
    ExampleString(int length, char fill);
    ~ExampleString();

    int Length() const { return fLength; }
    operator const char*() const { return fString; }
    unsigned Hash() const;
```

```

ExampleString& operator=(const ExampleString& s);
int operator==(const ExampleString& s) const;
int operator!=(const ExampleString& s) const;

const ExampleString operator+(const ExampleString& s) const;
ExampleString& operator+=(const ExampleString& s);

const ExampleString operator()(int first, int last) const;
                                Continued
char operator[](int position) const;

friend ostream& operator<<(ostream& stream, const ExampleString& s);

protected:

    int fLength;           // string length
    char *fString;         // storage allocated for the string
} ;

#endif

```

### 4.3 SOURCE FILE (CLASS IMPLEMENTATION) (\*.CC)

#### 4.3.1 ORDER OF CONTENTS

The items contained within a source file should be presented in the following order.

```

CM tool (ie. sccs) data
File Prolog (see 4.1)
#include < ... >
#include " ... "
#define ...
Declarations (for items applicable to this file only)
    Enumerations
    Typedefs
    Constants
    External Variables
    Globals
    Static Functions
Class Implementations (see 4.3.2)
Static Function Implementations (see 4.3.2)

```

#### 4.3.2 METHOD FUNCTION IMPLEMENTATION AND DOCUMENTATION IN SOURCE FILE

In the interest of reliability and maintainability of documentation, Code 522 has decided to maintain all method/function information in the source file only. Although this goes against the object oriented philosophy by requiring users of a class to view the source file, it is felt that the benefits in terms of accuracy and singular maintenance of information are more important.

Projects may optionally choose to use a script to copy the information from the source file to the header file, perhaps only at the time of each software delivery. A sample script can be found in Appendix A.

This description block is used in the source file, and is interspersed with the actual code as follows:

```

//--- Method Prolog -----
// Name:
// class::method
//
// Description:
// Description of the method written for a user or maintainer of the method
//
// Arguments:
// Name          Description
//
// Name          Description
//
// Etc.
//
// Returns:
// Type          Description
//
// Notes: Anything relevant about this class, including document references,
// assumptions, constraints, restrictions, abnormal termination conditions, etc.
//--- End Method Prolog -----
-----
Method or function declaration
Method or function implementation

//--- Method Prolog -----
//--- End Method Prolog-----
-----
Method or function declaration
Method or function implementation

Etc.

```

### 4.3.3 SAMPLE SOURCE FILE

```

static const char sccs[] = "%W% %G% (SCCS Control)";
/ / = = =                               F i l e                               P r o l o g
=====
//   This code was developed for NASA, Goddard Space Flight Center, Code 520
//   for the Style Guide Project.
//
//--- Contents -----
//
//   ExampleString class
//
//--- Description -----
//
//   A generic string class with the following members:
//       char* fString
//           A pointer to a dynamically allocated character array. Only
//           enough space is allocated to hold the value with an ending
// null.
//           If the string is null (length of 0), this pointer is set to
//           the constant gNullString, rather than just allocating a single
//           byte on the heap.
//       int fLength
//           The current length of the string, not counting the ending
//           null. If this is 0, it means the string is null, and that
//           fString points to the constant gNullString.
//
//--- Notes -----
//
//   This code has been shamelessly pilfered from the DstdString code to provide
//   a reasonable example of code written to the DSTD style guide.
//
//--- Development History -----
//
//   96/05/13          S. Developer/522
//   Initial version created to provide style guide example.
//
//--- Warning -----
//
//   This software is property of the National Aeronautics and Space Administration.
//   Unauthorized use or duplication of this software is strictly prohibited.
//   Authorized users are subject to the following restrictions:
//   · Neither the author, their corporation, nor NASA is responsible for
//   any consequences of the use of this software.
//   · The origin of this software must not be misrepresented either by
//   explicit claim or by omission.
//   · Altered versions of this software must be plainly marked as such.
//   · This notice may not be removed or altered.
//
/ / = = =                               E n d                               P r o l o g
=====

#include <string.h>

#include "ExampleString.h"

static char* gNullString = ""; // constant character array used to represent a null
string
                                // fString will be set to point to this if fLength is 0

```

```

//--- Method Prolog -----
// Name:
// ExampleString::ExampleString()
//
// Description:
// Construct a null string.
//
//--- End Method Prolog-----

(Continued)

ExampleString::ExampleString()
:
    fLength(0),
    fString(gNullString)
{
}

//--- Method Prolog -----
// Name:
// ExampleString::ExampleString(const ExampleString& s)
//
// Description:
// Copy constructor.
//
// Arguments:
// s                An existing string to duplicate
//
//--- End Method Prolog -----

ExampleString::ExampleString(const ExampleString& s)
:
    fLength(0),
    fString(gNullString)
{
    fLength = s.fLength;
    if (fLength)
    {
        fString = new char[fLength+1];
        strcpy(fString, s.fString);
    }
}

//--- Method Prolog -----
// Name:
// ExampleString::ExampleString(const char* p)
//
// Description:
// Construct a string with the value of a null-terminated character
// array. A null pointer will be accepted and treated as an empty
// string.
//
// Arguments:
// p                A null-terminated character array
//
//--- End Method Prolog-----

ExampleString::ExampleString(const char* p)
:

```

```

    fLength(0),
    fString(gNullString)
{
    // replace a null pointer with the null string
    if (!p)
    {
        p = gNullString;
    }
    fLength = strlen(p);
    if (fLength > 0)
    {
        fString = new char[fLength+1];
        strcpy(fString, p);
    }
}

(Continued)

//--- Method Prolog -----
// Name:
// ExampleString::ExampleString(int length, char fill)
//
// Description:
// Construct a string with a given number of fill characters.
//
// Arguments:
// length                The length of the string to create
//
// fill                  The character to be used to fill the DstdString
//
//--- End Method Prolog-----
ExampleString::ExampleString(int length, char fill)
:
    fLength(0),
    fString(gNullString)
{
    fLength = length < 0 ? 0 : length;
    if (fLength)
    {
        fString = new char[fLength+1];
        for (int i = 0; i < fLength; I++)
        {
            fString[i] = fill;
        }
        fString[fLength] = '\0';
    }
}

//--- Method Prolog -----
// Name:
// ExampleString::~~ExampleString()
//
// Description:
// Destructor deallocates any memory held by the string.
//
//--- End Method Prolog -----
ExampleString::~~ExampleString()
{
    // if fLength is not 0, fString must have been allocated, so delete it
    if (fLength)
    {

```

```

        delete[] fString;
    }
}

//--- Method Prolog -----
-----
// Name:
// ExampleString::Hash() const
//
// Description:
// Calculates a hash code for the string.
//
// Notes:
// This hashing function has not been certified.
//
//--- End Method Prolog -----
-----

                                (Continued)
unsigned ExampleString::Hash() const
{
    unsigned hash = 76123;

    for (const char* p = fString; *p; p++)
    {
        hash = (hash + *p) * 12377;
    }
    return(hash);
}

//--- Method Prolog -----
-----
// Name:
// ExampleString::operator=(const ExampleString& s)
//
// Description:
// Assign the value of an existing string to this one.
//
// Arguments:
// s                An existing string
//
// Returns:
// DstdString&      The value of the assignment
//
//--- End Method Prolog-----
-----
ExampleString& ExampleString::operator=(const ExampleString& s)
{
    // check for self assignment
    if (this == &s)
    {
        return *this;
    }

    if (fLength)
    {
        delete[] fString;
        fString = gNullString;
    }

    fLength = s.fLength;
    if (fLength)

```

```
{  
    fString = new char[fLength+1];  
    strcpy(fString, s.fString);  
}  
  
return(*this);  
}
```

```

                                (Continued)

//--- Method Prolog -----
// Name:
// ExampleString::operator+(const ExampleString& s)
//
// Description:
// Return the result of appending another string to this one.
//
// Arguments:
// s                An existing string
//
// Returns:
// const ExampleString    The concatenation of the two strings
//
//--- End Method Prolog-----
const ExampleString ExampleString::operator+(const ExampleString& s) const
{
    return(*this + s.fString);
}

//--- Method Prolog -----
// Name:
// ExampleString::operator+=(const ExampleString& s)
//
// Description:
// Set the value of this string to the result of appending another
// string to it.
//
// Arguments:
// s                An existing string
//
// Returns:
// ExampleString&    The modified string
//
//--- End Method Prolog-----
ExampleString& ExampleString::operator+=(const ExampleString& s)
{
    return(*this = *this + s);
}

//--- Method Prolog -----
// Name:
// ExampleString::operator==(const ExampleString& s)
//
// Description:
// Tests for equality with another string.
//
// Arguments:
// s                An existing string
//
// Returns:
// int              If 1, the strings have the same value; 0 - otherwise
//
//--- End Method Prolog -----
int ExampleString::operator==(const ExampleString& s) const
{

```

```

        return(strcmp(fString, s.fString) == 0);
    }

                                                                    (Continued)
//--- Method Prolog -----
// Name:
// ExampleString::operator!=(const ExampleString& s)
//
// Description:
// Tests for inequality with another string.
//
// Arguments:
// s                An existing string
//
// Returns:
// int              If 1, the strings do not have the same value; 0 - otherwise
//
//--- End Method Prolog -----
int ExampleString::operator!=(const ExampleString& s) const
{
    return(strcmp(fString, s.fString) != 0);
}

//--- Method Prolog -----
// Name:
// ExampleString::operator()(int first, int last) const
//
// Description:
// Return the substring of this string specified by the first and last
// character positions. If the first position is greater than or
// equal to the last position, a null string will be returned.
//
// Arguments:
// first            Zero based offset of the first character of the substring
//
// last            Zero based offset of the last character of the substring
//
// Returns:
// const ExampleString The specified substring
//
//--- End Method Prolog -----
const ExampleString ExampleString::operator()(int first, int last) const
{
    // Truncate the bounds if they are too high or too low.
    if (first < 0)
        first = 0;
    if (last >= fLength)
        last = fLength-1;

    int length = last - first + 1;
    if ((length <= 0) || (first > fLength))
        return "";

    ExampleString subString(length, ' ');
    strncpy(subString.fString, &fString[first], length);

    return(subString);
}

```



```

(Continued)
//--- Method Prolog -----
// Name:
// ExampleString::operator[](int position) const
//
// Description:
// Get the character at a given position.
//
// Arguments:
// position          The zero based offset of the character to return
//
// Returns:
// char              The specified character
//
//--- End Method Prolog -----
char ExampleString::operator[](int position) const
{
    if ((position < 0) || (position >= fLength))
    {
        return('\0');
    }
    return fString[position];
}

//--- Method Prolog -----
// Name:
// friend operator<<(ostream& stream, const ExampleString& s)
//
// Description:
// Puts a string on a stream.
//
// Arguments:
// stream            The output stream
//
// Returns:
// ostream&          The stream being put to
//
//--- End Method Prolog -----
ostream& operator<<(ostream& stream, const ExampleString& s)
{
    return(stream << s.fString);
}

```

## SECTION 5 USE OF LANGUAGE CONSTRUCTS

### 5.1 FUNCTIONS

- a. A **single return statement** at the end of a function creates a single, known point that is passed through at the termination of function execution. **Multiple returns** in a single unit should be avoided as much as possible, unless the use of a single return would cause the code to be difficult to understand or maintain.
- b. Using an expression (including a constructor call) in a return statement is more efficient than declaring a local variable and returning it (avoiding a copy-constructor and destructor call). Overdoing its use, however, increases the difficulty of debugging.
- c. Always declare the **return type** of functions. Do not default to integer type (int). If the function does not return a value, then give it return type void.
- d. In the case of a **thrown exception**, the code should attempt to leave a consistent state before throwing the exception (this includes deallocating memory that would otherwise be unreachable if the execution of the function should discontinue, and setting appropriate state variables, as necessary).

### 5.2 VARIABLES

- a. **Align** internal variable declarations (using tabs) so that the first letter of each variable name is in the same column.
- b. Declare each internal (local) variable on a separate line followed by an **in-line comment**. Loop indices and other such insignificant variables are an exception to this rule. They can all be listed on the same line with one comment.
- c. **Local variables** should be declared at the level at which they are needed. For example, if a variable is used throughout the procedure, it should be declared at the top of the procedure. If a variable is used only in a computational block, it may be declared at the top of that block. Local variable declarations should be commented well, so that they “stand out” for the code-reader/maintainer. Local variables should be **initialized** when they are declared.
- d. All variables should be described with an **in-line comment** to the right of their declaration.
- e. Do not use internal-variable declarations that override declarations at higher levels; these are known as **hidden variables**.
- f. In method and function declarations, all arguments must have a type and a name.
- g. In implementation, only arguments which are used within a method or function should be named.
- h. Floating point numbers should have at least one number on each side of the decimal point:
 

0.5	5.0	1.0e+36
-----	-----	---------
- i. Hexadecimal numbers should use 0x (zero, lower-case x) and upper case A-F:
 

0x123	0xFF
-------	------
- j. For portability, if the length of a variable is important, use a **typedef**. If length is not an issue, use int, which is normally selected to be the most efficient in terms of performance for a given platform.
- k. **Pass primitives** by value if the function will not change values. Pass primitives by reference if the function will change values.

1. **Pass objects** by constant reference if the function will not change the object. Pass by reference if the function will change the object.

### 5.3 CONSTANTS

There are several ways to specify constants: `const` modifier, `#define`, and enumeration data types.

- a. Avoid the use of the **#define** preprocessor command whenever possible. Use the **const** modifier instead of the `#define` preprocessor to define simple constants. Using `#define` can prevent type checking by the compiler.

Example 5.3a - Potential Problems with Using `#define`

```
#define kSize 10 + 10                                //10 + 10 will be substituted for ksize
...
area = kSize * ksize;                                //this evaluates to 10 + 10 * 10 + 10
                                                    //which is 10 + (10 * 10) + 10 = 120
```

Example 5.3b - Using the 'const' Modifier is Preferred

```
const int kSize = 10 + 10;                            // ksize evaluates to the number 20
...
area = kSize * kSize;                                // this evaluates to 20 * 20 = 400
```

- b. Enumeration types create an association between constant names and their values. By using this method (as an alternative to `#define`), constant values can be generated or values can be assigned. Place one variable identifier per line and use aligned braces and indentation to improve readability. When assigning values, align the values in the same column.

Example 5.3c - Enumeration Types

```
enum StackOperationResult
{
    kFull      = -2,
    kBadStack  = -1,
    kOkay     = 0,
    kNotEmpty  = 0,
    kEmpty     = 1
};
```

#### 5.4 POINTERS AND STRUCTURES

- a. The use of classes is preferred to structures; however, in some cases structures may be needed. Do not place methods within structures. If methods are needed within a structure, the structure should become a class.
- b. Set pointers to NULL after delete to guard against a “**double delete**” unless they are being reassigned immediately.

#### 5.5 TYPE CONVERSIONS, CASTS, AND VIRTUAL FUNCTIONS

- a. Type conversions occur by default when different types are mixed in an arithmetic expression across an assignment operator. Use the cast operator to make type conversions **explicit** rather than implicit. Avoid casting on pointers as shown in example 5.5a.

Example 5.5a - Dangerous Casting

```
class A
{
    ...
} ;

class B
{
public:
    ...

    void Function();
} ;

void Dangerous ()
{
    A a;

    B* pb = (B*) &a;    // This is bad.
    pb->Function();      // Function() doesn't exist in A.
}                       // Unknown result.
```

- b. This kind of cast is most commonly used when several different classes exist and you want to access a specific method of an object depending on the class of the object. A better way of accomplishing this is to use virtual functions, as in the following example.

## Example 5.5b - Use of Virtual Functions

```
class Base
{
    virtual void Do() = 0;
} ;

class Alpha : public Base
{
    ...
    virtual void Do() { DoAlpha(); }
    void DoAlpha();
} ;

class Beta : public Base
{
    ...
    virtual void Do() { DoBeta(); }
    void DoBeta();
} ;

void Foo::Bar (Base* base)
{
    base->Do();
}
```

If A and B are descendants of some class Base, and a method takes an object of type Base as an argument, you could determine if the argument was really of type A or B (using a Type field or some other technique), then call an A method or B method by first casting the pointer down.

- c. Place the pointer qualifier (\*) with the variable name rather than with the type. For example:  
*char \*s, \*t, \*u;*

## 5.6 USE OF OPERATORS

- a. Use **side-effects** within expressions sparingly. No more than one operator with a side-effect (=, op=, ++, --) should appear within an expression. It is easy to misunderstand the rules for C++ compilation and get side-effects compiled in the wrong order. The following example illustrates this point:  
*if ((a < b) && (c==d++))*

If a is not less than b, d will not be incremented.

Avoid using side-effect operators within relational expressions. Even if the operators do what the author intended, subsequent reusers may question what the desired side-effect was.

- b. Use the **comma operator** exceedingly sparingly. One of the few appropriate places is in a for statement. For example:  
*for (i = 0, j = 1; i < 5; i++, j++)*

## 5.7 ASSIGNMENT OPERATORS AND INITIALIZATION

- a. Objects should be initialized using parentheses (). For example:  
*DstdString x("woof");*

Primitives should be initialized using an equals sign (=). For example:  
*int x = 3;*

- b. C++ is an expression language. In C++, an assignment statement such as "a=b" itself has a value that can be embedded in a larger context as shown in example 5.7a. For maintainability, this feature should be avoided; however, it may be needed in some cases to improve performance. Example 5.7b illustrates the relative readability of nonembedded vs. embedded assignments.

Example 5.7a - Useful Embedded Assignment

```
while ((link = iter()) != NULL)
{
    if (link->Valid())
    {
        link->Write(cout);
    }
    else
    {
        link->Write(cerr);
        bad.Append(link);
    }
}
```

## Example 5.7b - Embedded Assignment Alternatives

not recommended:

```

if ((total = GetTotal()) == 10)
{
    cout << "goal achieved\n";
}

```

recommended:

```

total = GetTotal();
if (total == 10)
{
    cout << "goal achieved\n";
}

```

**5.8**      **CONDITIONAL EXPRESSIONS**

- a. In C++, conditional expressions allow evaluation of expressions and assignment of results in a **shorthand way**. For example, the following if-then-else statement could be expressed using a conditional expression as follows.

## Example 5.8a - Simple Conditional Expression

not recommended:

```

z=0;
if (a > b)
    z = a;
else
    z = b;

```

recommended:

```

z = (a > b) ? a : b;

```

- b. While some conditional expressions seem very natural, others do not, and their use should be avoided. The following expression, for example, is not as readable as the one above and would not be as easy to maintain. It should be broken into individual statements.

## Example 5.8b - Complex Conditional Expression (not recommended)

```

c = (a == b) ? d + f(a) : f(b) - d;

```

- c. Do not use **complex conditional expressions** if the algorithm can easily be expressed in a more clear, understandable manner. If conditional expressions are used, comments should be provided to aid the reader's understanding.

## 5.9 SELECTION CONTROL STATEMENTS (IF-THEN-ELSE)

- a. Indent one tab and use braces in the following format. If only one statement is contained within the block, **braces** should be added to improve clarity, following the format below:

```
if (condition)
{
    one statement;
}
else if (condition)
{
    another statement;
}
else
{
    statement 1;
    statement 2;
    ...
    statement n;
}
```

- b. Use **nested if statements** if there are alternative actions (i.e., there is an action in the else clause), or if an action completed by a successful evaluation of the condition has to be undone. Do not use nested if statements when only the if clause contains actions.

### Example 5.9a - Nested 'if' Statements

recommended nesting:

```
status = DeltaCreate((Callback) NULL, &delta);
if (status == kNdbOk)
{
    if (((status = DeltaRecordCondition(...)) == kNdbOk)
        && ((status = DeltaFieldCondition(...)) == kNdbOk)
        && ((status = DeltaFieldCondition(...)) == kNdbOk))
    {
        status = DeltaCommit(...);
    }
    NdbDestroyDelta(delta);
}
```

inappropriate nesting:

```

status = DeltaCreate((Callback) NULL, &delta);
if (status == kNdbOk)
{
    status = DeltaRecordCondition(...);
    if (status == kNdbOk)
    {
        status = DeltaFieldCondition(...);
        if (status == kNdbOk)
        {
            status = DeltaFieldCondition(...);
            if (status == kNdbOk)
            {
                status = DeltaCommit(...);
            }
        }
    }
    NdbDestroyDelta(delta);
}

```

- c. Because the else part of an if-else statement is optional, omitting the ‘else’ from a nested if sequence can result in ambiguity. Therefore, always use braces to avoid confusion and to make certain that the code compiles the way it was intended. In the following example, (5.9c) the same code is shown both with and without braces. The first example will produce the results desired. The second example will not produce the results desired because the ‘else’ will be paired with the second ‘if’ instead of the first.

#### Example 5.9b - Use of Braces

recommended:

```

if (n > 0)
{
    for (i = 0; i < n; i++)
    {
        if (s[i] > 0)
        {
            status = function();
            return(i);
        }
    }
}
else // CORRECT -- braces force proper association
{
    cout << "error - n is zero \n";
}

```

not recommended:

```

if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0)
        {
            status = function();
            return(i);
        }
    else // WRONG -- the compiler will match to closest else-less if
        cout << "error - n is zero \n";

```

## 5.10 SWITCH STATEMENTS

- a. For readability, use the following format for switch statements:

```
switch (expression)
{
case aaa:
    statement[s]
    break;
case bbb:    // fall through
case ccc:
    statement[s]
    break;
default:
    statement[s]
    break;
}
```

- b. Note that the **fall-through feature** of the C++ switch statement should be commented for future maintenance.
- c. All switch statements should have a **default case**, which may be merely a "fatal error" exit. However, for enumerated types there may be benefits to not using default in that the compiler may flag unspecified cases. The default case should be last and does not require a break, but it is a good idea to put one there anyway for consistency.

## 5.11 ITERATION CONTROL STATEMENTS

- a. 'While' statements should be of the following format:

```
while (expression)
{
    statement_1;
    ...
    statement_n;
}
```

- b. 'For' statements should be of the following format:

```
for (expression)
{
    one_statement;
}
for (expression)
{
    statement_1;
    ...
    statement_n;
}
```

- c. Do-while statements should be of the following format:

```
do
{
    statement_1;
    statement_2;
    statement_3;
}
while (expression);
```

- d. If only one statement is contained within the block, braces should be used to improve clarity.

## 5.12 GOTO'S AND LABELS

- Do not use gotos or labels.

## 5.13 MEMORY ALLOCATION

- Use new and delete rather than malloc and free.

## 5.14 STANDARD TYPEDEF SCHEME

It is recommended that the following 522 standard typedefs be used to improve portability and maintainability.

Fixed Size Fields:

typedef <one signed byte>	Byte;
typedef <one unsigned byte>	Ubyte;
typedef <two signed bytes>	Int16;
typedef <two unsigned bytes>	UInt16;
typedef <four signed bytes>	Int32;
typedef <four unsigned bytes>	UInt32;
typedef <eight signed bytes>	Int64;
typedef <eight unsigned bytes>	UInt64;

Macros for Cross Platform Development:

```
OsName <operating system name>
OsMajorVersion <operating system major version>
OsMinorVersion <operating system minor version>
OsTeenyVersion <operating system teeny version>
```

Platform Specific Macros:

```
SunOS
HPUX
Irix
```

## SECTION 6

### DESIGN AND IMPLEMENTATION

#### 6.1 GUIDELINES FOR PORTABILITY

- a. Consider detailed optimizations only on computers where they prove necessary. Optimized code is often obscure. Optimizations for one computer may produce worse code on another. Document code that is obscure due to performance optimizations and isolate the optimizations as much as possible.
- b. Some code/functions are inherently nonportable. If possible, organize source files so that the computer-independent code and the computer-dependent code are in separate files. That way, if the program is moved to a new computer, it will be clear which files need to be changed for the new platform.
- c. Different computers have different word sizes. If code relies on a (predefined) type being a certain size (e.g., int being exactly 32 bits), then a new type should be created (e.g., typedef long int32) and used (int32) throughout the program; further changes will require only changing the new type definition.
- d. Note that pointers and integers are not necessarily the same size; nor are all pointers necessarily the same size on various machines. Use the system function sizeof( ... ) to get the size of a variable type instead of hard-coding it.
- e. Beware of code that takes advantage of two's complement arithmetic. In particular, avoid optimizations that replace division or multiplication with shifts.
- f. Become familiar with the standard library and use it for string and character manipulation. Do not reimplement standard routines. Another person reading the code might see the reimplementation of a standard function and would need to establish whether that version does anything unique.
- g. Use #ifdefs to conceal nonportable quirks by means of centrally placed definitions.
- h. If a set of declarations is likely to change when code is ported from one platform to another, put those declarations in a separate header file.

Example 6.1a - Centrally Placed Definitions

```
#ifdef decus
#define UNSIGNED_LONG    long
#else
#define UNSIGNED_LONG    unsigned long
#endif
```

## 6.2 GUIDELINES FOR PERFORMANCE

- a. If performance is not an issue, then write code that is easy to understand instead of code that is faster. For example,

replace:

```
d = (a = b + c) + r;
```

with:

```
a = b + c;
```

```
d = a + r;
```

- b. When performance is important, as in real-time systems, use techniques to enhance performance. If the code becomes "tricky" (i.e., possibly unclear), add comments to aid the reader.
- c. Minimize the number of opens and closes and I/O operations if possible.
- d. Free allocated memory as soon as possible.
- e. To improve efficiency, use the automatic increment ++ and decrement operators -- and the special operations += and \*= (when side-effect is not an issue).
- f. When passing a structure to a function, use a reference. Using references to structures in function calls not only saves memory by using less stack space, but it can also boost performance. The compiler does not have to generate as much code for manipulating data on the stack and it executes faster.
- g. If arguments do not need to be modified, pass them to a function by const reference.

## 6.3 PROGRAM DESIGN LANGUAGE

- a. Program Design Language (**PDL**) is recommended as a means for generating and reviewing source code design prior to implementation. PDL should be English-like, and should be understandable to someone who is not an expert in the particular programming language. However, it is useful for the PDL to use actual variable and method names.
- b. It is recommended that PDL not be maintained during or after the implementation phase. It should evolve into code and comments.
- c. Use of actual class and variable names is encouraged in PDL to remove the ambiguities inherent in the English language. The following is an example of PDL that is both understandable and precise:

## Example 6.3a - Understandable PDL

```

Retrieve the time increments from fOpenIncrement and fCloseIncrement
Retrieve the selected item from the display list
Extract the associatedTswSet from the fSetList
For each subset in the associatedTswSet
    Extract the tsw's contained in this subset
    For each tsw in this subset
        Determine if the tsw Update will cause overlaps
        If yes and this is the first time overlaps were detected
            Display a dialog asking the operator whether the overlaps should be deleted
                If the operator asks to delete overlaps
                    Call the appropriate TswSet method to do this.
            Else
                Terminate the time increment/decrement operation
        End If
    End if
    If no overlaps were detected or the operator indicated that no overlaps
        are to be deleted
        Update this tsw's open time using the open increment
        Update this tsw's close time using the close increment
    End if
End for
End for
Redisplay the set display list

```

**6.4 COMPILER WARNINGS**

No warnings are acceptable in delivered code. The following table lists specific unacceptable compiler warnings. At a minimum, the developer should enable the compiler to perform checks for these conditions.

Unused variables
Unused arguments
Extra commas in enumerations
Non-void functions that do not return a value
Possible loss of data in automatic type conversion (ex. conversion from 'long' to 'short')
Use of uninitialized variables
Out-of-order initializer list in constructor
Variable name hiding (ex. local variable declared with same name as global variable)
Bad pointer conversions (ex. from char* to int*)

**APPENDIX A**

Sample perl script to copy method prologs from the source file to the header file:

```
#!/usr/bin/perl
# ===== F i l e P r o l o g =====
#
#   This code was developed for NASA, Goddard Space Flight Center, Code 520
#   for the Style Guide Project.
#
#----- Description -----
#
#   A perl script for extracting method prologs from a file.
#
#----- Options -----
#
#   Usage: <name> [-s|d|v] [options] [files]
#
#   Options:
#
#       -i<file>          input (source?) file
#       -o<file>          output (header?) file [modified]
#       -x<ext>           output extension
#       -e                edit files in place
#       -s                silent
#       -d                debug
#       -v                verbose
#
#----- Notes -----
#
#   Method prologs fitting the format of the Code 520 Style Guide
#   Version 2 are located in an input file and written to an output
#   file sorted by alphabetical order.
#
#   The format of method prologs is:
#
#       //--- Method Prolog -----
#       //      Name:
#       //      Class::Method (<parameters>)
#
#   followed by comment lines (beginning with //),
#
#   and are terminated by a line
#
#       //--- End Method Prolog -----
#
#   Several alternatives for the Name: field are recognized.
#
```

```

                                (Continued)

#----- Development History -----
#
#       96/06/05           B. Wiegand/522
#       Initial version created for style guide.
#
#----- Warning -----
#
#       This software is property of the National Aeronautics and Space
#       Administration.  Unauthorized use or duplication of this software
#       is strictly prohibited.  Authorized users are subject to the following
#       restrictions:
#           *           Neither the author, their corporation, nor NASA is responsible
#           *           for any consequences of the use of this software.
#           *           The origin of this software must not be misrepresented
#           *           either by explicit claim or by omission.
#           *           Altered versions of this software must be plainly marked
#           *           as such.
#           *           This notice may not be removed or altered.
#
# = = = = = E n d F i l e P r o l o g
# =====

$kPrologs="prologs";
$kPrologDelimiter="//-----
    -----\n";
%gPrologs;

$_Silent = 0;
$_Debug = 0;
$_Verbose = 0;

$_EditInPlace;
$_OutputExtension;
$_ExplicitInputFile;
$_ExplicitOutputFile;

# streams
$_INPUT = "_InputStream";
$_OUTPUT = "_OutputStream";
$_ERROR = STDERR;
$_REPORT = STDOUT;
$_DEBUG = 0;
$_VERBOSE = 0;

if ( $_Verbose ) {
    $_VERBOSE = $_REPORT;
    $_DEBUG = $_REPORT;
}

elseif ( $_Debug ) {
    $_DEBUG = $_REPORT;
}

if ( $_ExplicitInputFile ) {

```

```

        &Dcf($_ExplicitInputFile);
    }

                                (Continued)

&ProcessArglist();
&ProcessExplicit();
&Execute();

sub Process {
    local($ifile, $input, $output) = @_ ;
    &Report("Process> filename $ifile");

    $_StateDepth = 0;

    if ( $ifile =~ /\.+\.c/ ) {
        &SourceFile($&);
    }
    else {
        &Report("ignoring file $ifile");
    }
}

sub SourceFile {
    local($pattern) = @_ ;

    &Debug("getting prologs from $pattern");

    %gPrologs = ();
    local($prolog) = 0;
    local($name);
    local($key);
    local($lcount) = 0;

line:
    while (<$_INPUT>) {
        ++$lcount;

        # print "prolog $prolog, name $name, key $key\n";
        if ( ! $prolog ) {
            if ( m|^/-- Method Prolog ---| ) {
                $prolog = 1;
                $name = 0;
            }

            next line;
        }

        if ( m|^/-- End| ) {
            $prolog = 0;
            next line;
        }

        if ( ! $name ) {
            if ( m|^//s*Name:\s*(.+)\s*$| ) {
                $name = 1;
                $key = $1;
                StartProlog($key);
            }
        }
    }
}

```

```

        }
        elsif ( m|^//\s*$| ) {
        }

        (Continued)

elsif ( m|^//\s*Name:| ) {
    }
    elsif ( m|^//\s*(.+)\s*$| ) {
        $name = 1;
        $key = $1;
        StartProlog($key);
    }
    else {
        print "ERROR> no name\n";
        &Die("missing Name in method prolog: line $_");
    }

    next line;
}

if ( m|^//.*| ) {
    $gPrologs{$key} .= $_;
}
else {
    &Die("non comment in prolog");
}

}

foreach $key (sort keys(%gPrologs)) {
    print $_OUTPUT "$gPrologs{$key}";
}

print $_OUTPUT "$kPrologDelimiter";
}

sub StartProlog {
    local($key) = @_;
    &Report("StartProlog($key)");
    $gPrologs{$key} .= "$kPrologDelimiter";
    $gPrologs{$key} .= "//  Name: $key\n";
}

#      Code copied from dcf.pl to make method prolog extractor stand-alone
# Process switches
sub ProcessArglist {
    while ($ARGV[0] =~ /^~/) {
        local($a) = shift(@ARGV);
        &Debug("ProcessArglist> got _${a}_");

        if ( $a =~ /^~i(.+)/ ) {
            &Die("Multiple input files selected") if $_ExplicitInputFile;
            $_ExplicitInputFile = 1;
        }
        elsif ( $a =~ /^~o(.+)/ ) {
            &Die("Already selected in place editing") if $_EditInPlace;
            &Die("Already selected extension") if $_OutputExtension;
        }

        $_ExplicitOutputFile;
        $_ExplicitOutputFile = 1;
    }
}

```

```

        elseif ( $a =~ /^-x(.+)$/ ) {
            &Die("Already selected output file") if $_ExplicitOutputFile;

$_OutputExtension;

                                (Continued)

$_OutputExtension = $1;
    }
    elseif ( $a =~ /^-e$/ ) {
        &Die("Already selected output file") if $_ExplicitOutputFile;
        $_EditInPlace = 1;
    }
    elseif ( $a =~ /^-s$/ ) {
        &Die("silent and verbose are incompatible") if $_V
$_Debug;
        $_Silent = 1;
    }
    elseif ( $a =~ /^-v$/ ) {
        &Die("silent and verbose are incompatible") if $_Silent;
        $_Verbose = 1;
    }

    elseif ( $a =~ /^-d$/ ) {
        &Die("silent and debug are incompatible") if $_Silent;
        $_Debug = 1;
        $_Verbose = 1;
    }
    else {
        &Die("Unrecognized switch: $a");
    }
}

}

sub DisplayArglist {
    local($stream) = @_;
    print $stream "last argv index is $#ARGV\n";
    local($a);
    foreach $a (@ARGV) {
        print $stream "arg _${a}_\n"
    }
}

sub ShowGlobals {
    local($stream) = @_;

    print $stream "silent $_Silent\n";
    print $stream "debug $_Debug\n";
    print $stream "verbose $_Verbose\n";

    print $stream "edit in place $_EditInPlace\n";
    print $stream "output extension $_OutputExtension\n";
    print $stream "explicit input file $_ExplicitInputFile\n";
    print $stream "explicit output file $_ExplicitOutputFile\n";
}

sub Execute {
    local($file);
    foreach $file (@ARGV) {
        &Dcf($file);
    }
}

```

```

}

sub ProcessExplicit {
    if ( $_ExplicitInputFile ) {

        (Continued)

        &Dcf($_ExplicitInputFile);
    }
}

sub Dcf {
    local($input) = @_;
    local($ext) = "$$";

    &Report("Dcf($input)");

    while ( -e "DCF.$input.$ext" ) {
        ++$ext;
        &Debug("extending temporary search to $ext");
    }

    local($temporary) = "DCF.$input.$ext";

    # open the input and output streams

    open($_INPUT, $input)
        || &Die("unable to open input file $input: $!");

    open($_OUTPUT, ">$temporary")
        || &Die("unable to open temporary file $temporary: $!");

    &Process($input, $_INPUT, $_OUTPUT);

    close($_INPUT);
    close($_OUTPUT);

    &PostProcess($input, $temporary);
}

# post processing cleans up the files

sub PostProcess {
    local($input, $temporary) = @_;

    local($output) = "dcf.out";

    if ( $_ExplicitOutputFile ) {
        &Report("Explicit output file $_ExplicitOutputFile");
        $output = $_ExplicitOutputFile;
    }
    elsif ( $_EditInPlace ) {
        if ( $_OutputExtension ) {
            &Report("Backing up input $input to $input$_OutputExtension");
            &Rename($input, "$input$_OutputExtension");
        }
        else {
            &Report("Discarding original $input");
            &Unlink($input);
        }
    }
}

```

```

        $output = $input;
    }
    elseif ( $_OutputExtension ) {
        (Continued)
$output = "$input$_OutputExtension";
    }
    else {
        $output = "$input.dcf";
    }

    if ( -e $output ) {
        &Report("Output file $output exists, appending $temporary");
        &AppendFile($temporary, $output);
        &Unlink($temporary);
    }
    else {
        &Report("Renaming $temporary to $output");
        &Rename($temporary, $output);
    }
}

sub Die {
    local($message) = @_;
    die "dcf: $message\n";
}

sub Report {
    local($data) = @_;
    if ( ! $_Silent ) {
        print $_REPORT "$data\n";
    }
}

sub Error {
    local($data) = @_;
    print $_ERROR "$data\n";
}

sub Debug {
    local($data) = @_;
    if ( $_Debug ) {
        print $_DEBUG "$data\n";
    }
}

sub Verbose {
    local($data) = @_;
    if ( $_Verbose ) {
        print $_VERBOSE "$data\n";
    }
}

sub Rename {
    local($old, $new) = @_;
    rename($old, $new)

```

```
        || &Die("Rename $old $new failed");
    }

    sub Unlink {

        (Continued)
        local($toast) = @_ ;
        unlink($toast)
        || &Die("Unlink $toast failed");
    }

    sub AppendFile {
        local($appendage, $appendee) = @_ ;

        open(AFOUTPUT, ">>$appendee")
        || &Die("AppendFile> unable to open $name for appending: $!");
        open(AFINPUT, "$appendage")
        || &Die("AppendFile> unable to open $name for input: $!");

        while (<AFINPUT) {
            print AFOUTPUT $_;
        }

        close(AFOUTPUT);
        close(AFINPUT);
    }
}
```

## BIBLIOGRAPHY

Cargill, Tom C++ Programming Style, Reading, MA: Addison-Wesley, 1992.

Coplien, James O. Advanced C++ Styles and Idioms, Reading, MA: Addison-Wesley, 1992.

Doland, J. et. al., C Style Guide, SEL-94-003, Software Engineering Laboratory Series, Goddard Space Flight Center, August 1994.

Lapin, J.E. Portable C and Unix System Programming, Englewood Cliffs, NY: Prentice Hall Software Series, 1987.

Meyers, Scott Effective C++: 50 Specific Ways to Improve Your Programs and Designs. Reading, MA: Addison-Wesley, 1992.

Murray, Robert B. C++ Strategies and Tactics, Reading, MA: Addison-Wesley, 1992.

Shoan, W. et. al., C++ Style Guide: A Proposed Supplement to the "C Style Guide" draft, Goddard Space Flight Center, June 1995.

Software and Automation Systems Branch, C++ Style Guide, Version 1.0, Goddard Space Flight Center, July 1992.

Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++, Reading, MA: Addison-Wesley, 1994.